UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Simulation von Farbspritzern in VR

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von

Steffen Kutscher

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Bastian Krayer (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im April 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

In no field of computer science has the hardware developed as rapidly as in the field of computer graphics. Today, we can display complex, geometric scenes in real time in immersive systems and also integrate elaborate simulations.

The aim of this work is to realize the simulation of paint splashes in a virtual world. For this purpose, an application will be implemented with the help of Unity, that uses three different techniques to color the environment with the help of paint splashes. Based on this application, the limits and possibilities of the techniques in virtual environments are examined more closely.

This examination shows that an inverse projection produces the best results.

Zusammenfassung

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt wie im Bereich der Computergrafik. So können wir heute komplexe, geometrische Szenen in Echtzeit in immersiven Systemen darstellen und auch aufwendige Simulationen integrieren.

Ziel dieser Arbeit ist es, die Simulation von Farbspritzern in einer virtuellen Welt zu realisieren. Hierzu wird mithilfe von Unity eine Anwendung umgesetzt, die drei verschiedene Techniken verwendet, mit denen die Umgebung mithilfe von Farbspritzern eingefärbt werden kann. Auf Basis dieser Anwendung werden die Grenzen und Möglichkeiten der Techniken in virtuellen Umgebungen genauer untersucht.

Diese Untersuchung zeigt, dass eine inverse Projektion die besten Ergebnisse vorweist.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das dynamische Einfärben der virtuellen Welt bietet eine interessante Möglichkeit, mit der Umgebung zu interagieren. Spiele wie Splatoon oder Portal 2 zeigten bereits, dass dies möglich ist. In letzterem macht dies sogar den Großteil des Spiel-Prinzips aus.

Ebenso hat sich die VR-Technologie, um in virtuelle Welten einzutauchen, in den letzten Jahren enorm weiterentwickelt.

Es würde ein neues Level der Immersion erreicht werden, wenn es möglich ist, VR und eine färbbare Welt zu verbinden. Mit diesem Thema wird sich die Arbeit beschäftigen. Dazu werden iterativ drei verschiedene Ansätze umgesetzt, die aufeinander aufbauen, jedoch je nach Anwendungsfall unterschiedliche Vorteile haben.

Der erste Ansatz beschäftigt sich mit sogenannten Decals, eigenständigen Objekten, welche die Farbspritzer darstellen.

Der zweite Ansatz versucht, mithilfe von Raycasts die Nachteile des ersten Ansatzes auszubessern. Dazu wird eine Textur verwendet, die über die gesamte Welt gelegt wird, um darauf aufgetragene Farbspritzer auf Oberflächen zu blenden.

Der dritte Ansatz wird schließlich einige Nachteile des zweiten Ansatzes beheben, indem die Raycasts gegen eine inverse Projektion getauscht werden, da diese die gleiche Aufgabe performanter und optisch ansprechender erfüllt als Raycasts.

Besonderer Wert wird bei der Evaluierung der Ansätze auf die Performance gelegt, da diese in VR besonders wichtig ist, um Spielern ein realitätsnahes Erlebnis zu ermöglichen. Das bedeutet, das Spiel soll möglichst ohne Einbrüche der Bildrate laufen, dabei aber ein akzeptables Maß an Ästhetik bieten. Die Evaluierung wird daher in Vor- und Nachteile pro Ansatz gegliedert und anschließend verglichen.

2 Grundlagen

2.1 Unity

Spiele-Engines bilden das Grundgerüst für jedes Video-Spiel. Sie verwalten die Grafik- und Soundausgabe, berechnen Physik und vieles mehr. Die ersten kommerziellen Spiele-Engines wurden zunächst für ein bestimmtes Spiel entwickelt und später veröffentlicht. Das bekannteste Beispiel hierfür ist die Unreal Engine von Epic Games, die ursprünglich für das Spiel Unreal entwickelt wurde. [1, S. 1]

Unity ist eine von Unity Technologies entwickelte Spiele-Engine. In Unity verwendet man sogenannte *GameObjects* (oder einfach Objekte), die als Container für viele verschiedene Arten von Komponenten dienen. Diese Komponenten weisen Unity an, mit den Objekten auf vordefinierte Weise zu interagieren. So kümmert sich Unity beispielsweise selbst um die grafische Darstellung, wenn ein Objekt eine Renderer-Komponente besitzt, und um die Berechnung von Physik, wenn eine Rigidbody-Komponente vorhanden ist. In diesen Komponenten können weitere Einstellungen zum Verhalten vorgenommen werden, so bietet die Rigidbody-Komponente Regler für Masse, Reibung, Gravitation und mehr. Des weiteren kann man einem Objekt ein oder mehrere Skript-Komponenten hinzufügen. Diese Skripts, die sich in Unity der Programmiersprache C# bedienen, bieten Schnittstellen, die von Unity zu bestimmten Zeitpunkten aufgerufen werden, beispielsweise beim Programmstart oder pro Frame.

2.2 Unity's Partikelsystem

Bei einem Partikelsystem handelt es sich um eine Technik, viele Objekte, sogenannte Partikel effizient zu erstellen und anzuzeigen. Bei diesen Partikeln handelt es sich um Positionen mit relativ wenigen individuellen Eigenschaften. So können diese im Partikelsystem angegeben und auf jeden Partikel übertragen werden. Flüssigkeiten, Rauch, Wolken, Flammen und ähnliche Phänomene werden häufig mithilfe von Partikelsystemen simuliert.

Unity bietet diese Funktionalität durch eine *Particle System*-Komponente. Diese besteht aus vielen Modulen, in denen sich etliche Einstellungen vornehmen lassen. Im Basis-Modul kann man zum Beispiel Farbe, Geschwindigkeit und Rotation festlegen, und im *Collision*-Modul kann man einstellen, womit die Partikel kollidieren können. Im Basis-Modul gibt es zudem eine Einstellung namens *Simulation Space*, die festlegt, ob die Partikel relativ zum Ursprung der Welt oder zum Partikelsystem berechnet werden. In dieser Arbeit wird stets die Einstellung *World* verwendet.

Insbesondere das *Renderer*-Modul ist wichtig für die Darstellung der

Partikel, denn hier wird festgelegt, welcher *Render Mode* verwendet wird, um die Partikel anzuzeigen. Normalerweise benutzt man hier die Einstellung *Billboard*, die bewirkt, dass der Partikel bei der Anzeige immer zur Kamera gedreht ist, wodurch ein räumliches Volumen simuliert wird. Außerdem bestimmt man hier ein Material, das die Textur für alle Partikel vorgibt.

2.3 Renderer

Renderer bezeichnen in Unity Komponenten, die dafür verantwortlich sind, Objekte anzuzeigen. Hier werden alle Materialien eingetragen, aus denen das Objekt besteht. Zudem kann man hier individuelle Einstellungen zur Belichtung vornehmen, also ob beispielsweise das Objekt Schatten wirft, selbst Schatten abbekommen kann und ob es bei der statischen Lichtberechnung einbezogen wird.

2.4 Shader

Auf modernen Grafikkarten gibt es einige fest verbaute Shadereinheiten. Nennenswert sind hier besonders Vertex- und Fragment-Shader. Ersterer erhält als Eingabe die Vertices der darzustellenden Szenerie, kann diese verändern und muss sie anschließend in Bildschirm-Koordinaten transformieren. Der Fragment-Shader erhält danach für jeden Pixel des Bildschirms interpolierte Werte der vom Vertex-Shader berechneten Vertices und berechnet dann die resultierende Anzeige-Farbe des Pixels. Diese Shadereinheiten werden mit Shader-Programmen (Shadern) programmiert und bieten so dem Entwickler die Möglichkeit, in die Anzeige einzugreifen, um die Beleuchtung und Ähnliches nach eigenen Wünschen anzupassen. [3, VL 8]

2.5 Colormap

In dieser Arbeit wird mehrfach eine Textur verwendet, die ähnlich einer Lightmap funktioniert, allerdings werden in dieser keine Informationen zur Beleuchtung gespeichert, sondern welche Farbe an der jeweiligen Stelle aufgetragen wurde. Der Leserlichkeit halber wird diese Textur fortan als **Colormap** bezeichnet.

Lightmaps sind Texturen, die mithilfe von zum Beispiel Raytracing die Belichtungsinformationen für jede statische Oberfläche speichern. [6, S. 2] Diese Informationen werden dann zur Render-Zeit wieder verfügbar gemacht, indem der Shader zu jedem Vertex passende Lightmap UV-Koordinaten bekommt.

2.6 Virtuelle Realität und HMD

Virtuelle Realität (VR) ist eine Technologie, mit welcher der Nutzer in speziell dafür entwickelte virtuelle Umgebungen eintauchen kann. Dabei werden die Sinne des Nutzers größtenteils von der Realität getrennt, indem sie mit Informationen aus der VR überdeckt werden. Insbesondere Sicht und Gehör müssen beeinflusst werden, um die Illusion aufrecht zu erhalten. Das Gehör wird mittels eines Headsets oder mit einem guten Surround Sound System stimuliert, während die Sicht mithilfe eines sogenannten HMD simuliert wird.

HMD steht für den englischen Term *head-mounted display*, also ein am Kopf angebrachter Bildschirm. Allerdings handelt es sich dabei um wesentlich mehr als eine simple Anzeige. Um den größten Teil des Sichtfeldes abzudecken, sind viele HMDs mit konvexen Linsen ausgestattet, welche die Sicht auf das Display weiter wirken lassen. Außerdem besitzen HMDs verschiedene Sensoren, die mit dem Computer kommunizieren, um die exakte Position und Rotation im Raum festzulegen. Dies wird dann übersetzt auf eine Position in der virtuellen Umgebung und aus dieser Perspektive kann nun ein Bild pro Auge generiert werden. Falls die Übersetzung von Bewegung oder Rotation zu langsam ist, kann dies unter Umständen zu Übelkeit oder Schwindel führen. [2]

2.7 Stand der Forschung: Splatoon und Portal 2

Das persistente Einfärben der Umgebung wurde bereits in einigen Spielen verwendet. Beispiele hierfür sind Splatoon und Portal 2.

Splatoon wurde von Nintendo für die Wii U entwickelt und veröffentlicht. Es ist ein Action-Shooter in Third-Person-Perspektive, bei dem der Spieler die Steuerung eines Inklings übernimmt, wodurch er sich in einen Tintenfisch verwandeln und schneller in Farbe schwimmen kann. In dem Spiel gibt es verschiedene Spiel-Modi, wobei für diese Arbeit besonders der Online-Mehrspieler-Modus relevant sein sollte. In diesem Modus haben die Spieler das Ziel, die Welt mit der Farbe des eigenen Teams einzufärben (siehe Abb. 1a). Hierzu steht dem Spieler eine Vielzahl an Waffen und Werkzeugen zur Verfügung, die auf verschiedene Arten Farbe verspritzen und dazu dienen, entweder Gegnern zu schaden oder die Umgebung einzufärben (siehe Abb. 1b). Es gewinnt das Team, das den größten Teil der Welt gefärbt hat. [1]

Portal 2 ist ein von Valve entwickeltes und veröffentlichtes Rätsel-Spiel

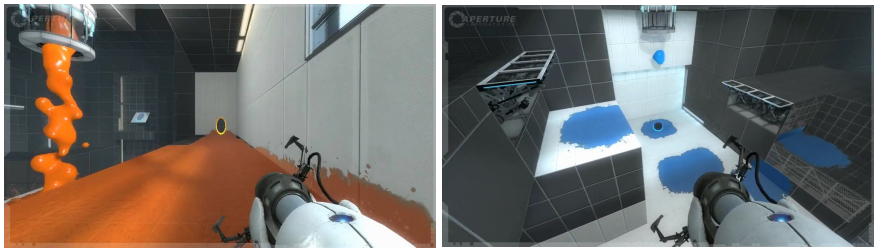
[1] https://www.wikiwand.com/de/Splatoon

[2] https://nintendo-online.de/wiiu/games/7777/splatoon/images/page:9

(a) Eingefärbte Umbegung **(b)** Der Klecksroller

Abbildung 1: Beispiele aus Splatoon [2]

in Ego-Perspektive, das auf Windows, Mac, Linux, Xbox 360 sowie Playstation 3 spielbar ist. Dem Spieler wird eine Waffe gegeben, mit der er zwei miteinander verbundene Portale platzieren kann. Mithilfe dieser Portal-Kanone wird er mit verschiedenen Testräumen sowie einigen Außenbereichen konfrontiert, in denen er versuchen muss, den Ausgang zu erreichen. Hierfür müssen meist Bodenplatten gedrückt oder Schalter betätigt werden. Neben der Portal-Kanone stehen dem Spieler zu diesem Zweck je nach Level Kugeln, Würfel oder andere Hilfs-Objekte zur Verfügung. Im Zusammenhang mit dieser Arbeit sind besonders die sogenannten Gels nennenswert, mit denen der Spieler Flächen färben kann (siehe Abb. 2). Diese verändern die Eigenschaften der Oberflächen. So verringert das orange-farbene *Propulsion-Gel* die Reibung, wodurch Objekte darauf leichter rutschen und Spieler sich sehr stark beschleunigen können. Das blaue *Repulsion-Gel* hingegen stößt Dinge ab, was Spielern ermöglicht, höher zu springen. Außerdem gibt es noch das weiße *Conversion-Gel*. Weiß gefärbte Flächen ermöglichen es dem Spieler, dort Portale zu platzieren, auch wenn das auf der Fläche sonst nicht möglich wäre. [3]



(a) Oranges Gel für höhere Beschleunigung [4] **(b)** Blaues Gel, um höher springen zu können [5]

Abbildung 2: Beispiele der Gels aus Portal 2

[3] https://www.wikiwand.com/de/Portal_2
[4] https://www.youtube.com/watch?v=Pcf99_DZZew
[5] https://www.youtube.com/watch?v=a7V0HBwHfEw

3 Prototypische Implementierung

Zur Implementierung wird der Einfachheit halber die Spiele-Engine Unity verwendet.

Jede Szene der prototypischen Implementierung benutzt als Basis ein **Partikelsystem**, mit dem man Farbpartikel verschießen kann. Neben dem Renderer benötigt dieses Partikelsystem lediglich das *Collision*-Modul (siehe Abb. 3). Die wichtigste Einstellung hier ist das Häkchen bei *Send Collision Messages*, damit Kollisionen über eine Funktion im zugehörigen Skript verarbeitet werden können. Diese heißt *OnParticleCollision* und wird pro getroffenem Objekt in einem Frame maximal einmal aufgerufen. Daher müssen hier als erstes alle Partikel-Kollisionen mit diesem Objekt abgerufen werden, dies erfolgt mit einem Aufruf der Funktion *ParticlePhysicsExtensions.GetCollisionEvents*.[6] Anschließend werden die Partikel-Kollisionen an ein weiteres Skript zur Kollisionsbehandlung übergeben, das sich um die szenenabhängigen weiteren Schritte kümmert.

Abbildung 3: Collision-Modul des Partikelsystems

Weiterhin benötigt jede Szene ein **Virtual Reality** Setup. Hierzu wird das VRTK[7] verwendet, um die Einbindung in Unity zu vereinfachen. Um nun einen VR-Spieler zu erstellen, wird einem Objekt das *VRTK_SDKManager* Skript hinzugefügt. Nun wird ein passendes SDK importiert, in diesem Fall das SteamVR Plugin. Um das SDK zu verwenden, wird ein neues leeres Objekt erstellt, dem das *VRTK_SDKSetup* Skript hinzugefügt wird. Anschließend fügt man diesem Objekt das *[Camera Rig]* Prefab aus dem SDK als Unterelement hinzu. Um auch die Controller verwenden zu können, wird ein leeres Objekt pro Controller benötigt. Diese Objekte werden vom VRTK *Left/Right Controller Script Alias* genannt und müssen im *VRTK_SDKManager* Skript referenziert werden. Um nun verschiedene Funktionen für die Controller zu ermöglichen, werden den Alias-Objekten verschiedene Skripte aus dem VRTK hinzugefügt, insbesondere sind die Skripte *VRTK_ControllerEvents*, *VRTK_InteractTouch*, *VRTK_InteractGrab* und *VRTK_InteractUse* nötig, um mit anderen Gegenständen interagieren zu können.

[6] `https://docs.unity3d.com/ScriptReference/`
[7] `https://vrtoolkit.readme.io/`

Zum Verschießen der Farb-Partikel dient in VR eine einfache Pistole, die den Beispielen des VRTK entnommen wurde. An diese wird das zuvor beschriebene Partikelsystem als Unterelement angehängt. Dieses sendet immer dann Partikel aus, wenn der Spieler den Abzug des Controllers betätigt.

3.1 Decals

Die erste Szene verwendet ein weiteres Partikelsystem, um sogenannte Decals in der Welt darzustellen. Inspiriert wurde diese Szene durch ein Unity Tutorial. [5]

Decal „A picture or design printed on special paper, that can be put onto another surface, such as metal or glass“.[8]
Ein Decal bezeichnet in der Computergrafik eine Textur, die auf Oberflächen aufgetragen werden kann, um beispielsweise Effekte wie Einschusslöcher oder Blutspritzer darzustellen.

Um diesen Effekt zu erzielen, wird ein neues Partikelsystem in die Szene eingefügt. Dieses benötigt keine besonderen Module bis auf den *Renderer*. Wichtig ist hier bloß, dass der *Render Mode* auf *Mesh* gesetzt und ein passendes Mesh, in diesem Fall ein *Quad* ausgewählt wird, da die Partikel nicht immer der Kamera zugewandt sein sollen. Als Material bekommt das Partikelsystem die Decal Textur zugewiesen. Auf die *Quads* wird dadurch automatisch die Decal Textur aufgetragen und in der Welt platziert (siehe Abb. 4).

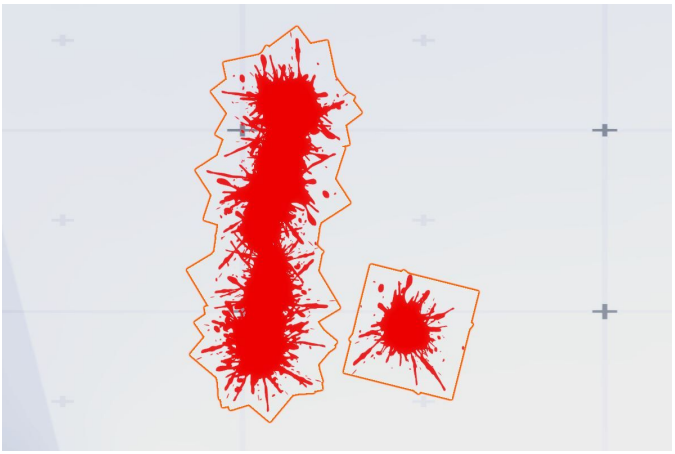


Abbildung 4: Decals mit sichtbaren *Quads*

[8] `https://dictionary.cambridge.org/us/dictionary/english/decal`

Bei der Kollision eines Partikels mit einer geeigneten Fläche wird nun eine Funktion aufgerufen, die ein neues Decal-Partikel aus Kollisionspunkt und Normale der Oberfläche sowie zufälligen Werten für Rotation und Größe generiert. Anschließend wird dieses in der Decal-Liste gespeichert. Mithilfe dieser Liste wird nun das Decal-Partikelsystem aktualisiert.

3.1.1 Vorteile

Decals bieten die Möglichkeit, detaillierte Texturen zu verwenden, um die Farbspritzer darzustellen. Dabei bleiben Ränder deutlich und Details auf den Texturen erhalten.

Zudem ermöglichen die zufällige Rotation und Größe trotz der Verwendung einer einzigen Spritzer-Textur die Simulation einer hohen optischen Variation, was eine bessere Immersion für den Spieler bietet.

3.1.2 Nachteile

Der gravierendste Nachteil dieser Implementierung ist das Überragen der Decals an konvexen Flächen oder Kanten (siehe Abb. 5).

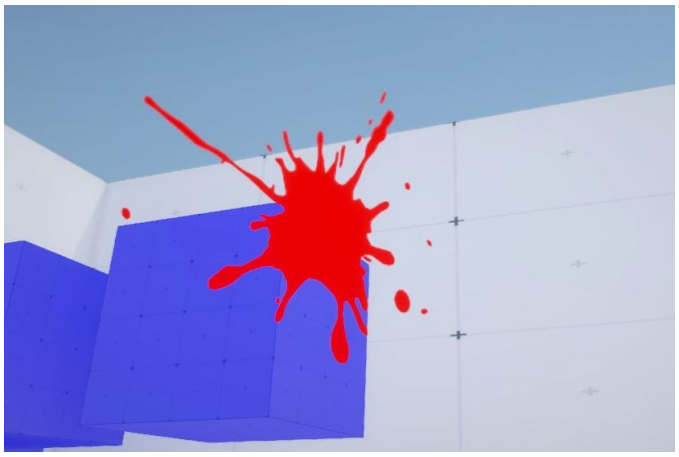


Abbildung 5: Nachteil von Decals

Ein weiteres Problem entsteht bei einer langen Spiel-Dauer, da jedes Decal gespeichert wird. Diese Daten müssen bei jedem neuen Decal an das Decal-Partikelsystem übergeben werden. Dadurch steigt der Rechenaufwand linear mit zunehmender Anzahl der Decals. Auf dem Testsystem (siehe Anhang A) sank die Bildrate auf 10 bis 20 Bilder pro Sekunde (FPS) bei etwa 10000 Decals. Dieses Problem kann dadurch umgangen werden, dass ein zirkulierendes Decal-Array verwendet wird. So werden die ältesten Decals überschrieben, sobald das Array voll ist. Allerdings kann man dann möglicherweise nicht mehr die gesamte Welt einfärben, da man auf eine zuvor festgelegte Anzahl an Decals beschränkt ist.

3.2 Raycasting auf Colormap

Die zweite Szene ist ein Versuch, die Nachteile der ersten Szene zu beheben. Hierzu wird zunächst ein einfacher Shader geschrieben, welcher mithilfe der UV-Koordinaten der Lightmap die Farben aus einer passenden Colormap ausliest. Diese Farbe wird dann über die Farbwerte der Oberfläche geblendet.

Die benötigte Colormap wird zu Szenenbeginn generiert, wobei jeder Pixel auf die Farbe *Clear*, also (0, 0, 0, 0) gesetzt wird. Bei Kollision eines Partikels mit einer färbbaren Oberfläche werden nun Raycasts in zufällige Richtungen um den Kollisionspunkt mit einer zuvor bestimmten maximalen Länge ausgesandt. Immer wenn ein Raycast auf eine färbbare Oberfläche auftrifft, wird an der passenden Stelle in der Colormap ein farbiger 3×3 Pixel großer Farbspritzer eingezeichnet, welcher dann dank des Shaders auf dem Objekt zu sehen ist (siehe Abb. 6).

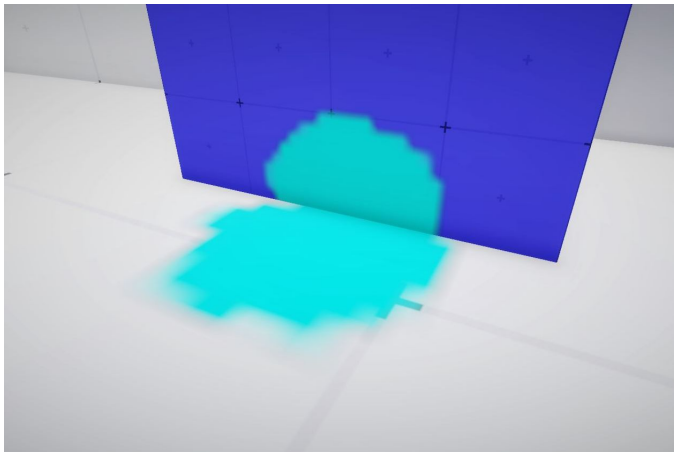


Abbildung 6: Spritzer, der aus 2500 Raycasts entstanden ist

Wichtig zu beachten ist hier, dass färbbare Oberflächen und Objekte statisch sein müssen, damit sie auf der Lightmap eingetragen werden und zugehörige UV-Koordinaten besitzen. Ansonsten gibt es keine passende färbbare Stelle in der Colormap.

3.2.1 Vorteile

Durch die Verwendung einer Colormap wird ein Überragen der Farbspritzer verhindert, da keine eigenständigen Objekte erstellt werden, auf die die Farbe aufgetragen wird. Zudem sorgen die Raycasts dafür, dass die Farbe sich um Ecken legt, da verschiedene Stellen um den Einschlagspunkt herum getroffen werden.

3.2.2 Nachteile

Raycasts: In dieser Implementierung wird zwar kein zusätzlicher Rechenaufwand pro Farbspritzer benötigt, wie bei den Decals, doch ist stattdessen viel Rechenleistung für die hohe Anzahl an Raycasts pro Frame erforderlich. Dies führt bei hoher Schussfrequenz in VR unter Umständen zu Verzögerungen oder sogar Aussetzern, sodass der Körper aus der eigenen Bewegung außerhalb des Spiels andere Informationen ableitet, als die Augen liefern. Dieser Sinneskonflikt kann Orientierungslosigkeit, Schwindel oder andere Symptome der sogenannten Cybersickness auslösen. [2, S. 52] Auf dem Testsystem führte eine hohe Schussrate zu Berechnungszeiten von 150 bis 200 Millisekunden, was einer Bildrate von 5 bis 6 FPS entspricht. Dieses Problem könnte man umgehen, indem man die Schussrate limitiert oder die Anzahl der Raycasts pro Kollision reduziert.

Ein weiterer Nachteil besteht darin, dass die Raycasts es nicht ermöglichen, eine Textur aufzutragen, da sie zufällig ausgesendet werden. Dadurch ist das Resultat stets relativ kreisförmig. Außerdem werden dank des zufälligen Aussendens der Raycasts einige Stellen der Colormap bei einem einzigen Schuss mehrfach bearbeitet.

Colormap: Die Qualität der Spritzer ist von der Auflösung der Colormap abhängig, die von der Leistung der Grafikkarte beschränkt wird.

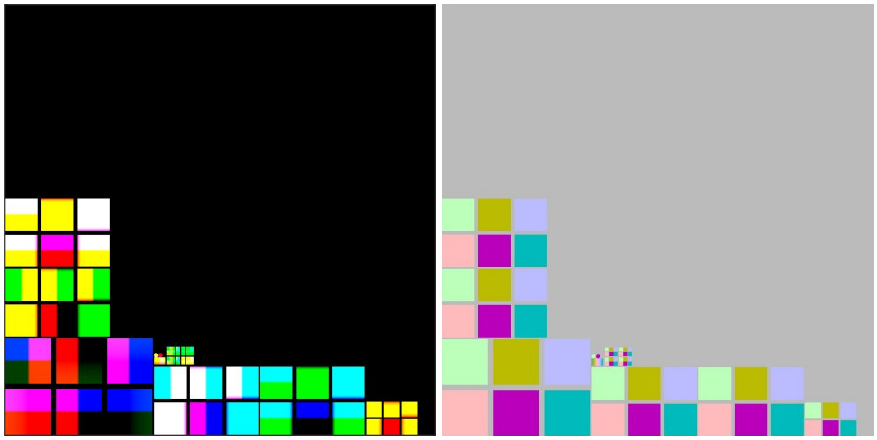
Zusätzlich ist es nicht möglich, dynamische Flächen einzufärben, da dann die UV-Koordinaten der Lightmap fehlen.

3.3 Inverse Projektion auf Colormap

In der dritten und letzten Szene wird versucht, die Vorteile der ersten beiden Ansätze zu kombinieren.

Wie zuvor wird das Prinzip einer Colormap verwendet. Nun werden beim Beginn der Szene drei Texturen pro Lightmap erstellt. Zum einen die Colormap, bei der jeder Pixel wie zuvor leer initialisiert wird. Zum anderen werden eine Welt-Positions-Map (Welt-Pos-Map) und eine Welt-Normalen-Map benötigt. Dazu wird die Welt aus der Perspektive einer orthographischen Kamera im Zentrum der Welt zweimal mit verschiedenen Shadern gerendert. Beide Shader setzen im Vertex-Shader die Position auf die zugehörigen UV-Koordinaten der Lightmap, damit die resultierende Map genauso angeordnet ist. Der Fragment-Shader für die Welt-Pos-Map gibt die ursprüngliche Welt-Position als Farbe zurück, während der andere Fragment-Shader die Normale als Farbe codiert. Da die Welt-Pos-Map detaillierte Informationen bereitstellen muss, wird das Format der Textur auf *ARGBFloat* initialisiert, damit die gespeicherten Werte nicht auf 32 Bit beschränkt sind. Damit an Rändern später keine Interpolations- oder Rundungsfehler auftreten, werden die Flächen in der Welt-Pos-Map

und der Welt-Normalen-Map anschließend noch jeweils um ein Pixel erweitert. Somit haben wir nun drei Texturen, die uns anhand der Lightmap UV-Koordinaten die Welt-Position, die Normale dieser Fläche und die aufgetragene Farbe ermitteln lassen.



(a) Welt-Positions-Map **(b)** normierte Welt-Normalen-Map

Abbildung 7: Generierte Maps

Bei Kollision eines Partikels mit einer färbbaren Oberfläche wird diesmal eine Matrix berechnet, die das Inverse zur Model-Matrix der Kollision bildet. Mithilfe dieser Matrix lässt sich bestimmen, ob ein Punkt, der in Welt-Koordinaten gegeben ist, im Bereich der Projektion liegt. Diese Matrix, die Textur für den Spritzer und dessen Farbe werden nun an einen sogenannten Blit-Shader übergeben, der den Spritzer auf die Colormap auftragen soll. Da dieser Shader auf die Colormap angewandt wird, erhält er zunächst keine Informationen über die zugehörigen Objekte. Dazu bekommt er Zugriff auf die zuvor generierten Texturen mit Welt-Positionen und Normalen.

Der Vertex-Shader übergibt dem Fragment-Shader nun die interpolierte Lightmap UV-Koordinate, die als Zugriffsschlüssel für die Welt-Pos-Map, Welt-Normalen-Map und Colormap dient. Die Welt-Position wird dann mit der inversen Model-Matrix in das Projektions-Koordinatensystem überführt. Falls diese Welt-Position nun von der Projektion betroffen ist, wird noch geprüft, ob die Normale der Projektion zugewandt ist. Sind beide Kriterien erfüllt, wird die übergebene Farbe mit dem Alpha-Wert der Farbspritzer-Textur auf die Colormap übertragen. Falls dort bereits eine andere Farbe vorhanden war, werden beide mit dem Porter-Duff-Algorithmus zusammengemischt. [4, S. 256]

Zum Auftragen der Colormap auf die Umgebung wird derselbe Shader verwendet wie in der zweiten Szene (siehe Abb. 8). [7]

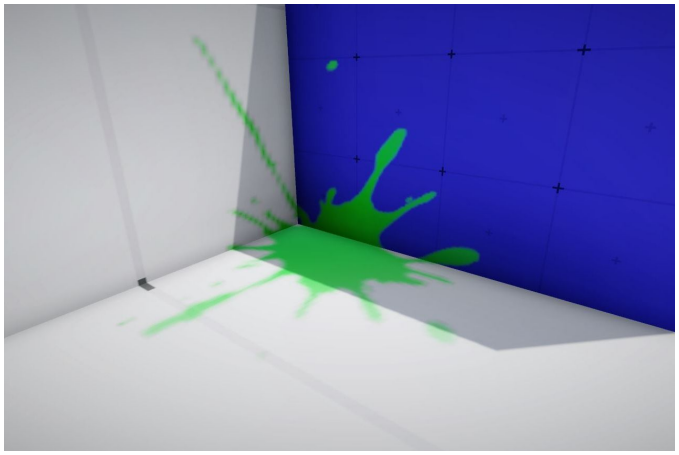


Abbildung 8: Spritzer, der sich dank inverser Projektion um Ecken legt

3.3.1 Vorteile

Diese Implementierung behebt alle vorigen Probleme, die durch Raycasts entstanden sind. Nun muss die CPU nur noch die inverse Model-Matrix bestimmen. Dadurch kommt es nicht mehr zu Aussetzern. Auf dem Testsystem blieb die Bildrate konstant bei 110 bis 120 FPS. Außerdem wird durch die inverse Projektion eine Farbspritzer-Textur abgetastet, wodurch keine kreisförmigen Spritzer mehr entstehen müssen.

3.3.2 Nachteile

Die unter 3.2.2 aufgeführten Nachteile, die aus der Verwendung einer Colormap resultieren, bleiben weiterhin vorhanden.

3.4 Vergleich der Implementierungen

Die Decals benötigen zur eigentlichen Erstellung am wenigsten Rechenleistung, da nur bereits vorhandene Werte verwendet werden. Position sowie Normale an der Kollisionsfläche werden vom Partikelsystem bei der Kollision mitgeliefert, und Rotation und Größe sind Zufallswerte. Somit ist das Bestimmen der Decal-Partikel nicht besonders anspruchsvoll. Doch der Rechenaufwand steigt proportional zur Anzahl an erstellten Decals, da jedes erstellte Decal zusammen mit allen alten Decals an das Partikelsystem übergeben werden muss. Die Belastung der GPU verhält sich ähnlich, da immer mehr Decals gerendert werden müssen. Auf dem Testsystem bot diese Implementierung zu Beginn eine Bildrate von über 100 FPS, bei vielen Decals allerdings sinkt diese auf 10 bis 20 FPS bei der Erzeugung der Decals.

Eigenschaft	Decals	Raycasts	inv. Projektion
Bildrate bei Decal-Erstellung	10-120 von Spieldauer abhängig	5-10 bei hoher Frequenz	110-120
CPU-Last	*Linear steigend* Übertragung an Partikelsystem	*Hoch* Hoher Rechenaufwand der Raycasts	*Gering* Berechnen einer inversen Matrix
GPU-Last	*Linear steigend* Anzahl zu rendernder Decals steigt	*Gering, konstant* Ein zusätzlicher Texturzugriff	*Konstant* Blit-Operationen
Aussehen	*Hübsche Textur* Gut für gerade Flächen, schlecht an Kanten	*Keine Textur* Zufällige Raycasts	*Gut* Textur verwendbar, legt sich um Ecken

Tabelle 1: Vergleich der Implementierungen

Der grafische Aufwand der Raycast-Methode ist wesentlich konstanter, da beim Rendern der Umgebung lediglich ein weiterer Texturzugriff pro Fragment erfolgt. Diese ist proportional zur Lightmap bemessen und ändert sich somit zur Laufzeit des Programms nicht weiter. Allerdings erfordern Raycasts wesentlich mehr Rechenleistung als Decals. Pro Kollision werden etwa 2000 Raycasts benötigt, damit die entstehende Fläche möglichst wenige Löcher aufweist. Zudem wird die Colormap mit jedem Raycast bearbeitet. All dies geschieht über die CPU. Bei einer hohen Schussfrequenz zeigten sich auf dem Testsystem Bildraten von 5 bis 10 FPS.

Die inverse Projektion ist ein Kompromiss aus beiden vorherigen Ansätze. Sie benötigt mehr Rechenleistung als die Decals, da eine inverse Matrix bestimmt werden muss, allerdings ersichtlich weniger als 2000 Raycasts. Zudem erfordert sie mehr Grafik-Leistung als die Raycast-Methode, da die Projektion von einem Shader ausgeführt wird, der zusätzlich zum normalen Rendering aufgerufen wird. Doch ein zusätzlicher Render-Durchlauf über eine Textur stellt für VR-fähige Grafikkarten kein Problem dar. Da diese sich zur Laufzeit nicht mehr ändern, besteht auch hier keine Gefahr eines Problems im späteren Spielverlauf. Hier zeigten Versuche auf dem Testsystem, dass sich die Bildrate nicht ändert. Sie blieb konstant bei 110 bis 120 FPS.

Die inverse Projektion übertrifft die beiden anderen Ansätze hinsichtlich der optischen Darstellung. Sie ermöglicht die Verwendung einer Textur und passt sich an Ecken an. Lediglich die Auflösung der Colormap ist ein limitierender Faktor. Decals hingegen sehen zwar schön aus, aller-

dings funktionieren diese nur auf ebenen Oberflächen, ohne die Illusion der aufgetragenen Farbe zu zerstören. Die Raycast-Methode ist dahingehend schlechter, da stets eine kreisförmige Fläche entsteht.

4 Fazit

Zusammenfassend lässt sich daraus erkennen, dass die inverse Projektion für das Einfärben der Umgebung am geeignetsten ist, da das Aussehen und die Bildrate besonders gute Werte geliefert haben. Zudem bieten sich für ein nicht-permanentes Einfärben die Decals an, da diese im kleinen Maßstab performant und einfach zu implementieren sind. Hier muss nur beachtet werden, die Decals klein zu halten, damit der Effekt des Überragens an Kanten nicht zu offensichtlich ist. Raycasts sind in diesem Anwendungsfall für Echtzeitsimulationen leider nicht konkurrenzfähig.

5 Weitere Möglichkeiten

Während der Entwicklung der prototypischen Implementierung sowie dem Schreiben der Ausarbeitung entstanden einige Ideen, wie das Einfärben der Welt in VR weiter verbessert werden könnte. Allerdings gehen diese Ideen über den Umfang dieser Arbeit hinaus:

Die vorgestellten Ansätze weisen optische sowie physikalisch detailgetreue Mängel auf. Um die Optik der Farbspritzer weiter zu steigern, wäre es möglich, für die Colormap eine passende Normal-Map zu bestimmen, welche der Farbe mehr Substanz verleihen würde. Zudem ließe sich hier ein Rauschen hinzufügen, um die Oberfläche der Farbe unebener zu gestalten. Dadurch könnte man auch die möglicherweise niedrige Auflösung der Colormap ausgleichen, da dann der Aliasing-Effekt nicht mehr so stark zu erkennen ist.

Eine weitere Möglichkeit, das Erlebnis zu intensivieren wäre, eine physikalisch realistischere Simulation von Farbspritzern zu ermöglichen, indem man entweder die Raycast-Methode optimiert oder eine Flüssigkeits-Simulation implementiert, um zum Beispiel das Verfließen der Farbe nachzubilden oder Tropfen von der Decke zu ermöglichen. Ob dies allerdings mit der Leistung handelsüblicher Rechner echtzeitfähig wäre, bleibt unklar.

Des Weiteren kann man die Ansätze, die eine Colormap verwenden, dahingehend erweitern, dass auch nicht-statische Objekte, die also nicht auf der Lightmap vorhanden sind, färbbar sind, indem man diesen Objekten eine eigene Colormap hinzufügt, die über beispielsweise ein UV Unwrap des Objekts referenziert wird.

Falls die Welt auf mehrere Lightmaps verteilt ist und dementsprechend auch über mehrere Colormaps verfügt, kann die Performance der inversen Projektion weiter gesteigert werden. Dazu wird während dem Berech-

nen der inversen Model-Matrix mithilfe einer *Axis-Aligned Bounding Box* bestimmt, welche Objekte vom Farbspritzer getroffen werden. Nun kann vor der Anwendung des Blit-Shaders gefiltert werden, über welche Colormaps dieser iterieren muss.

Anhang

A Testsystem

Prozessor	Intel® Core™ i7-4790K (4 Kerne à 4 GHz)
Grafikkarte	NVIDIA GeForce GTX 970
Arbeitsspeicher	8 GB
Betriebssystem	Windows 10 Pro
HMD	HTC Vive (Gen 1) Kabelgebunden

Literatur

[1] Will Goldstone. *Unity Game Development Essentials*. Packt Publishing Ltd., Birmingham, 2009.

[2] Joseph J. LaViola Jr. A discussion of cybersickness in virtual environments. *ACM SIGCHI Bulletin*, 32(1):47–56, 2000.

[3] Stefan Müller. Vorlesung: Computergraphik I, 2017.

[4] Thomas Porter and Tom Duff. Compositing digital images. *ACM Siggraph Computer Graphics*, 18(3):253–259, 1984.

[5] Unity. Live Session: Controlling Particles Via Script, 2017.

[6] Michal Valient. *3D Engines in games - Introduction*. 2007.

[7] Michael Voeller. Splatoon in Unity, 2017.